

REMARKS

Claims 1, 3-9, 11-17, 19, and 20 are pending in the present application. Independent claims 1, 9, and 17 are amended to recite subject matter previously presented in claims 2, 10, and 18, respectively. Claims 3-5, 11-13, 19, and 20 are amended to correct dependency. Claims 5, 13, and 20 are amended to more clearly recite terminating the waiter thread if the queue is empty. Claims 6 and 14 are amended to recite shutting down the virtual machine in response to termination of the waiter thread. Support for these amendments can be found on at least page 12, lines 13-22, of the present specification. Reconsideration of the claims is respectfully requested.

I. 35 U.S.C. § 103, Obviousness

The Office Action rejects claims 1-20 under 35 U.S.C. § 103 as being unpatentable over JAVA Threads (by Scott Oaks & Henry Wong, Copyright 1999, by O'Reilly & Associates, Inc.). This rejection is respectfully traversed.

With respect to the independent claims, the Office Action states:

Per claims 1, 9, and 17, the reference of Oaks & Wong, teaches: A method for preventing premature shutdown of a virtual machine (pg. 164, Daemon Threads, to pg. 167, in particular, pg. 167, lines 1-2) comprising the steps of:

monitoring daemon threads running in the virtual machine by the waiter thread (page 167, first paragraph, Daemon thread acquire the lock and release the Daemon lock, once critical section is complete; See also page 199, Definition of Monitor and lock, therefore); and

preventing shutdown of the virtual machine so long as any of the monitored daemon threads are running in the virtual machine (we protect Daemon thread by ensuring that a user thread exists, As long as there is user thread, the virtual machine will not exist, which will allow the daemon threads to finish the critical section of the code the system of Oaks & Wong, page 167, lines 1-7).

The reference of Oaks & Wong fails to explicitly teach of “starting a waiter thread in the virtual machine”. However, daemon thread, which acquires the daemon lock, must have assurance from a user thread, in order for the virtual machine not to exit before daemon thread finishes its job. Therefore, it would have been obvious for one ordinary skill in the art at the time the invention was made to start a user thread in order to ensure the daemon thread has finished its job, for the reason to make sure all

daemon threads has finished their jobs. Also Oaks & Wong suggested this on page 166, lines 5-20.

Office Action, dated June 7, 2004. JAVA Threads teaches a “DaemonLock” class that protects daemon threads by ensuring that a user thread exists. The cited portion of JAVA Threads states:

Implementation of the DaemonLock class is simple: we protect daemon threads by ensuring that a user thread exists. As long as there is a user thread, the virtual machine will not exit, which will allow the daemon threads to finish the critical section of code. Once the critical section is completed, the daemon thread can release the daemon lock, which will terminate the user thread. If there are no other user threads in the program at that time, the program will exit. The difference, however, is that it will exit outside of the critical section of code.

JAVA Threads, page 167. Thus, JAVA Threads teaches that a daemon thread may start a DaemonLock thread, which is a user thread, when executing a critical section of code. The daemon thread may then release the daemon lock when the critical section of code is finished, thus terminating the DaemonLock user thread. In other words, each daemon thread must perform actions to protect itself from termination when the virtual machine shuts down.

As seen in the implementation of the DameonLock class on page 166 of JAVA Threads, a daemon thread calls the acquire() method when executing a critical section of code. The acquire() method increments a “lockCount” variable, sets the DaemonLock class to a non-daemon state, and starts the DaemonLock thread. When a daemon thread releases the lock, the daemon thread calls the release() method. The release() method decrements the lockCount variable. The DaemonLock thread runs and executes a wait() instruction as long as the lockCount variable is not zero. Thus, the DaemonLock thread ensures that a non-daemon thread runs whenever a daemon thread is executing a critical section of code.

In JAVA Threads, the DaemonLock thread only holds a counter that has a non-zero value if a daemon thread is executing a critical section of code. It is up to the daemon thread to notify the DaemonLock thread that a lock is needed by calling the acquire() method. In other words, in the JAVA Threads reference, daemon threads may

run without acquiring a lock. The DaemonLock class does not monitor daemon threads. In contradistinction, the present invention starts a wait thread and registers daemon threads in a queue managed by the waiter thread. The waiter thread then monitors all daemon threads registered in the queue. The present invention then prevents shutdown of the virtual machine so long as any of the monitored daemon threads are running in the virtual machine. JAVA Threads does not teach or suggest a waiter thread that maintains a queue and monitors daemon threads that are registered in the queue.

With respect to originally presented claims 2, 10, and 18, the Office Action states:

Per claims 2, 10 and 18 the method as recited in claim 1, further comprising the steps of:

registering daemon threads in a queue managed by the waiter thread. The reference of Oaks & Wong, page 213, teaches of queue busy flag, when lock is used to protect daemon thread, and in the art of computer science, elements inserted in the queue are registered first.

Office Action, dated June 7, 2004. Applicant respectfully disagrees. JAVA Threads does teach a queue busy flag and maintaining a queue when multiple threads attempt to lock a resource. However, JAVA Threads does not teach that the DaemonLock class operates as a resource. Rather, JAVA Threads expressly teaches that the DaemonLock class is locked by incrementing a lockCount variable. Therefore, the portion of JAVA Threads in chapter 8 that describes techniques for managing lock starvation is not at all related to the DaemonLock class described in chapter 6. Furthermore, there is no need in JAVA Threads for maintaining a queue for the DaemonLock class, because the lockCount variable keeps track of whether a daemon thread has acquired a lock on the DaemonLock thread.

While the JAVA Threads reference teaches a user thread that solves the same problem as the claimed waiter thread, the DaemonLock thread of JAVA Threads solves the problem in a substantially different manner. The waiter thread of the present invention maintains a queue of all running daemon threads and monitors the daemon threads in the queue. Therefore, the waiter thread of the present invention is advantageous because the code of the daemon threads need not be specifically programmed to acquire a lock when executing a critical portion of code and releasing a lock whenever critical jobs are completed. Furthermore, in JAVA Threads, daemon

threads may be running that have yet to begin processing critical portions of code. In that case, the virtual machine may shut down before a daemon thread even begins critical jobs.

The applied reference does not teach or suggest each and every claim limitation; therefore, JAVA Threads does not render claims 1, 9, and 17 obvious. Since claims 3-8, 11-16, 19, and 20 depend from claims 1, 9, and 17, the same distinctions between JAVA Threads and claims 1, 9, and 17 apply for these claims. In addition, claims 3-8, 11-16, 19, and 20 recite other additional combinations of features not suggested by the reference.

More particularly, with respect to claims 3-5, 11-13, and 19-20, the Office Action states:

Per claims 3, 11, and 19 the method as recited in claim 2, further comprising the step of:

responsive to a first daemon thread becoming inactive, searching for other inactive daemon threads registered in the queue, it is well known in the art that queues are data structure from which elements can be removed only in the same order they are were inserted. Therefore, when the first daemon thread is inactive, the other daemon threads are first in line (Oaks & Wong, page 213, last line to page 214).

Per claims 4 and 12, the method as recited in claim 2, further comprising the step of:

responsive to a new daemon thread being created, appending the new daemon thread to the end of the queue. It is well known in the art that queues are data structure from which elements can be removed only in the same order they are were inserted. The queue in Oaks & Wong accepts data sequentially.

Per claims 5, 13 and 20 the method as recited in claim 2, further comprising the steps of

determining whether the queue is empty; and if the queue is empty, waiting for the virtual machine to shut down or for a new daemon thread to be created (see Oaks & Wong, page 167, lines 1-7)

Office Action, dated June 7, 2004. Applicant respectfully disagrees. JAVA Threads does not teach or suggest using queues to lock the DaemonLock thread. Rather, JAVA Threads teaches using a lockCount variable to acquire a lock. Therefore, it would not have been obvious to a person of ordinary skill in the art to modify the teachings of JAVA Threads to search for inactive threads in a queue maintained by a waiter thread, to

append new daemon threads to the queue, or to determine whether the queue is empty, as in the claimed invention.

Furthermore, with respect to claims 6 and 14, the Office Action states:

Per claims 6 and 14, the method as recited in claim 5, further comprising the step of:

if a new daemon thread is created, adding the new daemon thread to the queue. When a daemon thread is created, as part of the scheduling scheme, it is added to the queue, so that when there is no user thread the system can find other threads before exiting, therefore it must be added to a data structure such as a queue (Oaks & Wong, page 165, lines 7-15).

Office Action, dated June 7, 2004. Applicant respectfully disagrees. The cited portion of JAVA Threads states:

The only time the Java virtual machine checks to see if particular threads are daemon threads is after a user thread has exited. When a user thread exits, the Java virtual machine checks to see if there are any remaining user threads left. If there are user threads remaining, then the Java virtual machine, using the rules we've discussed, schedules the next thread (user or daemon). If, however, there are only daemon threads remaining, then the Java virtual machine will exit and the program will terminate. Daemon threads only live to serve user threads; if there are no more user threads, there is nothing to serve and no reason to continue.

JAVA Threads, page 165, lines 7-15. Thus, JAVA Threads teaches that the Java virtual machine checks whether there are any **user** threads remaining. The only time the Java virtual machine checks whether any **daemon** threads remain is when there are no more user threads. In this case, there would be no waiter thread to maintain a queue. Therefore, it is unclear how this portion of JAVA Threads somehow teaches adding a new daemon thread to a queue, particularly because JAVA Threads teaches that the DaemonLock thread maintains only a lockCount variable, and not a queue.

Therefore, Applicant respectfully requests withdrawal of the rejection of claims 1-20 under 35 U.S.C. § 103.

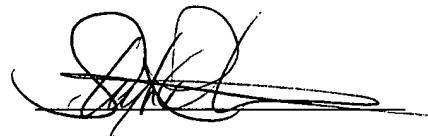
II. Conclusion

It is respectfully urged that the subject application is patentable over the prior art of record and is now in condition for allowance.

The Examiner is invited to call the undersigned at the below-listed telephone number if in the opinion of the Examiner such a telephone conference would expedite or aid the prosecution and examination of this application.

DATE: Sep 30, 2004

Respectfully submitted,



Stephen R. Tkacs
Reg. No. 46,430
Yee & Associates, P.C.
P.O. Box 802333
Dallas, TX 75380
(972) 367-2001
Agent for Applicant